TRAINING CUỐI KÌ 2 - K17

Cấu trúc dữ liệu và giải thuật

Trainer

BÙI HỮU NGHĨA
TRẦN THẢO QUYÊN

CHƯƠNG 2: TÌM KIẾM VÀ SẮP XẾP

2.1. CÁC GIẢI THUẬT TÌM KIẾM

2.1.1. Tìm kiếm tuyến tính (Linear Search)

2.1.1*. Tìm kiếm tuyến tính (cải tiến)

2.1.2. Tìm kiếm nhị phân (Binary Search)

2.2. CÁC GIẢI THUẬT SẮP XẾP

2.2.1. Phương pháp chọn trực tiếp (Selection Sort)

2.2.2. Phương pháp chèn trực tiếp (Insertion Sort)

2.2.3. Phương pháp vun đống (Heap Sort)

2.2.4. Phương pháp phân hoạch (Quick Sort)

2.2.5. Phương pháp trộn (Merge Sort)

CHƯƠNG 2: TÌM KIẾM VÀ SẮP XẾP

NỘI DUNG:

Tên thuật toán: Tiếng Anh, Tiếng Việt

1. Ý tưởng
2. Thuật toán
3. Các bước tiến hành
4. Cài đặt
5. Minh họa (Chạy tay từng bước)

2.1.1. TÌM KIẾM TUYẾN TÍNH (LINEAR SEARCH) (1)

Tên khác: Tìm kiếm tuần tự (Sequential Search)

1. Ý tưởng:

Duyệt toàn bộ danh sách A để xác định a[i] và trả về i nếu tồn tại a[i].

2.1.1. TÌM KIẾM TUYẾN TÍNH (LINEAR SEARCH) (2)

2. Thuật toán:

Input: Danh sách A chứa n phần tử có hoặc không có thứ tự, giá trị khóa x cần tìm

Output: Chỉ số i của phần tử a[i] trong A có giá trị khóa là x.

Trong trường hợp không tìm thấy i=-1

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(n)

Trung bình: O(n)

Trường hợp tốt nhất: O(1)

2.1.1. TÌM KIẾM TUYẾN TÍNH (LINEAR SEARCH) (3)

3. Các bước tiến hành:

B1: Khởi gán i=0;

B2: So sánh a[i] với giá trị khóa x cần tìm, có 2 khả năng:

+ (a[i]==x): tìm thấy x. Dừng

+ (a[i]!=x): sang bước 3;

B3: i=i+1; // Xét phần tử kế tiếp trong mảng

Nếu i==n: hết mảng, không tìm thấy x. Dừng

Ngược lại: lặp lại bước 2;

2.1.1. TÌM KIẾM TUYẾN TÍNH (LINEAR SEARCH) (4)

4. Cài đặt:

```
int LinearSearch(int a[], int n, int x) {
    int i = 0;
    while (i < n && a[i] != x)
        i++;
    if (i == n)
        return -1; //không tìm thấy x
    else
        return i; //tìm thấy x
    // trả về chỉ số i của phần tử a[i] trong A có giá trị khóa là x
}
```

2.1.1*. TÌM KIẾM TUYẾN TÍNH (CẢI TIẾN) (1)

1. Ý tưởng:

- Thêm phần tử a[n] có khóa x vào A, khi này A có n+1 phần tử. Phần tử thêm vào được gọi là phần tử cầm canh.
- Chỉ cần điều kiện dừng là tìm thấy phần tử a[i] có khóa x.

-> Rút gọn điều kiện dừng (thay vì mỗi lần lặp phải kiểm tra khóa và số lượng phần tử thì bây giờ chỉ cần kiểm tra khóa cần tìm)

2.1.1*. TÌM KIẾM TUYẾN TÍNH (CẢI TIẾN) (2)

2. Thuật toán:

Input: Danh sách A chứa n phần tử có hoặc không có thứ tự, giá trị khóa x cần tìm

Output: Chỉ số i của phần tử a[i] trong A có giá trị khóa là x.

Trong trường hợp không tìm thấy i=-1

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(n)

Trung bình: O(n)

Trường hợp tốt nhất: O(1)

2.1.1*. TÌM KIẾM TUYẾN TÍNH (CẢI TIẾN) (3)

3. Các bước tiến hành:

B1: Khởi gán i=0;

Khởi gán a[n]=x; // a[n] là phần tử cầm canh

B2: So sánh a[i] với giá trị khóa x cần tìm, có 2 khả năng:

+ (a[i]==x): sang bước 4;

+ (a[i]!=x): sang bước 3;

B3: i=i+1; // Xét phần tử kế tiếp trong mảng

B4: Nếu i<n: tìm thấy x. Dừng

Nếu i==n: vị trí tìm thấy là vị trí của phần tử cầm canh -> không tìm thấy x. Dừng.

2.1.1*. TÌM KIẾM TUYẾN TÍNH (CẢI TIẾN) (4)

4. Cài đặt:

```
int TimKiemTuyenTinhCaiTien(int arr[], int n, int x)
{
    int i = 0;
    arr[n] = x; // phần tử cầm canh
    while (arr[i] != x)
        i++;
    if (i < n)
        return i;
    else // i == n
        return -1;
}
```

2.1.2. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH) (1)

1. Ý tưởng:

- Chọn a[m] ở giữa A để tận dụng kết quả so sánh với khóa x.

A được chia thành 2 phần: trước và sau a[m]. Chỉ số bắt đầu, kết thúc của A là l, r.

- Nếu x = a[m], tìm thấy và dừng

- Xét thứ tự x, a[m]. Nếu thứ tự này

+ Là R, thì tìm x trong đoạn [l,r] với r=m-1;

+ Ngược lại, tìm x trong đoạn [l,r] với l= m+1;

2.1.2. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH) (2)

2. Thuật toán:

Input: Danh sách A chứa n phần tử đã có thứ tự R, giá trị khóa x cần tìm

Output: Chỉ số i của phần tử a[i] trong A có giá trị khóa là x.

Trong trường hợp không tìm thấy i=-1

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(log(n))

Trung bình: O(log(n))

Trường hợp tốt nhất: O(1)

2.1.2. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH) (3)

3. Các bước tiến hành:

B1: left=0; right=n-1;

B2:

- mid(left+right)/2;
- So sánh a[mid] với x. Có 3 khả năng:

+ a[mid]=x: tìm thấy x. Dừng

+ a[mid]>x: right=mid-1; // tìm tiếp x trong dãy con a[left]...a[mid-1]

+ a[mid]<x: left=mid+1; // tìm tiếp x trong dãy con a[mid+1]...a[right]

B3: Nếu left<=right: lặp lại bước 2

Ngược lại: không tìm thấy x. Dừng

2.1.2. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH) (4)

4. Cài đặt:

```
int BinarySearch(int arr[], int n, int x)
{
    int left = 0;
    int right = n - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (arr[mid] == x)
            return mid;
        if (x < arr[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

2.2.1. PHƯƠNG PHÁP CHỌN TRỰC TIẾP (SELECTION SORT) (1)

1. Ý tưởng:

Chọn phần tử nhỏ thứ i theo thứ tự R trong danh sách A và đặt vào vị trí i của danh sách

2.2.1. PHƯƠNG PHÁP CHỌN TRỰC TIẾP (SELECTION SORT) (2)

2. Thuật toán:

Input: A={a[0], a[1],…,a[n-1]} chưa có thứ tự R

Output: A={a[0], a[1],…,a[n-1]} đã có thứ tự R

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(n^2)

Trung bình: O(n^2)

Trường hợp tốt nhất: O(n^2)

2.2.1. PHƯƠNG PHÁP CHỌN TRỰC TIẾP (SELECTION SORT) (3)

3. Các bước tiến hành:

B1: Khởi gán i=0;

B2: Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[n-1]

B3: Đổi chỗ a[min] và a[i]

B4: Nếu i<n-1: i=i+1; Lặp lại bước 2;

Ngược lại: Dừng

2.2.1. PHƯƠNG PHÁP CHỌN TRỰC TIẾP (SELECTION SORT) (4)

4. Cài đặt:

```
void SelectionSort(int a[], int n) { // dãy tăng dần
    int min; // min là chỉ số phần tử nhỏ nhất trong dãy hiện hành
    for (int i = 0; i < n - 1; i++) {
        min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j; // lưu vị trí phần tử hiện nhỏ nhất
            }
        }
        swap(a[min], a[i]);
    }
}
```

2.2.2. PHƯƠNG PHÁP CHÈN TRỰC TIẾP (INSERTION SORT) (1)

1. Ý tưởng:

- Danh sách chỉ có 1 phần tử luôn có thứ tự. Như vậy, A_0={a[0]} là danh sách có thứ tự R
- Để sắp xếp A={a[0], a[1],…,a[n-1]} theo thứ tự R: lần lượt lấy a[i] (i>0) trong A và thực hiện:

+ Bắt đầu từ cuối danh sách A[i-1]={a[0],…,a[i-1]}, tìm vị trí k đầu tiên thỏa điều kiện a[k] R a[i]

+ Đẩy tất cả phần tử (nếu có) từ ngay sau vị trí k về bên phải 1 vị trí.

+ Đưa vào a[i] vị trí k+1 của A[i-1], A[i-1] thành A[i]

2.2.2. PHƯƠNG PHÁP CHÈN TRỰC TIẾP (INSERTION SORT) (2)

2. Thuật toán:

Input: A={a[0], a[1],…,a[n-1]} chưa có thứ tự R

Output: A={a[0], a[1],…,a[n-1]} đã có thứ tự R

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(n^2)

Trung bình: O(n^2)

Trường hợp tốt nhất: O(n)

2.2.2. PHƯƠNG PHÁP CHÈN TRỰC TIẾP (INSERTION SORT) (3)

3. Các bước tiến hành:

// Giả sử có đoạn a[0] đã được sắp

Bước 1: khởi gán i=1;

Bước 2: x=a[i]; // x là phần tử cần chèn

Bước 3: dời chỗ các phần tử từ a[pos] đến a[i-1] sang phải 1 vị trí để dành chỗ cho a[i]

Bước 4: a[pos]=x; // chèn vào đoạn {a[1]-a[i]} đã được sắp xếp

Bước 5: i=i+1;

Nếu i<n: Lặp lại bước 2;

Ngược lại: Dừng;

2.2.2. PHƯƠNG PHÁP CHÈN TRỰC TIẾP (INSERTION SORT) (4)

4. Cài đặt:

```
void InsertionSort(int a[], int n) {
    int pos, i;
    int x; //dùng lưu a[i] để tránh bị ghi đè khi dời chỗ các phần tử
    for (i = 1; i < n; i++) //đoạn a[0] đã sắp xếp
    {
        x = a[i];
        pos = i - 1;
        while ((pos >= 0) && (a[pos] > x)) // tìm vị trí chèn x
        {
            a[pos + 1] = a[pos]; //kết hợp dời chỗ các phần tử sẽ đứng sau x trong dãy mới
            pos--;
        }
        a[pos + 1] = x; // chèn x vào dãy
    }
}
```

2.2.3. PHƯƠNG PHÁP VUN ĐỐNG (HEAP SORT) (1)

1. Ý tưởng:

- Xây dựng cấu trúc Heap:

+ Là 1 cây nhị phân hoàn chỉnh

+ Nếu giá trị khóa của nút cha và 2 nút con lần lượt là K, K_1, K_2 thì:

$$\begin{cases} K_1 \ R \ K \\ K_2 \ R \ K \end{cases}$$

+ Nếu dùng mảng A để biểu diễn cấu trúc Heap, A có đặc điểm:

A[0] là cực đại theo R

A[i*2+1] R A[i] và A[(i+1)*2] R A[i]

A[i*2+1] và A[(i+1)*2] là phần tử liên đới với A[i]

- Sắp xếp dựa trên cấu trúc Heap:

+ Hoán đổi vị trí của A[0] và A[n-1], đưa giá trị cực đại về cuối dãy → dãy A trong bước tiếp theo đã giảm được một phần tử, còn lại là A[0]...A[n-2]

+ Bắt đầu từ A[0], điều chỉnh các phần tử trong A[0]...A[n-2] để đảm bảo tính chất của Heap.

+ Thực hiện hoán đổi A[0] và điều chỉnh dãy mới đến khi dãy A chỉ còn lại một phần tử

2.2.3. PHƯƠNG PHÁP VUN ĐỐNG (HEAP SORT) (2)

2. Thuật toán:

Input: A={a[0], a[1],…,a[n-1]} chưa có thứ tự R

Output: A={a[0], a[1],…,a[n-1]} đã có thứ tự R

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(nlogn)

Trung bình: O(nlogn)

Trường hợp tốt nhất: O(nlogn)

2.2.3. PHƯƠNG PHÁP VUN ĐỐNG (HEAP SORT) (3)

3. Các bước tiến hành:

Giai đoạn 1 : Hiệu chỉnh dãy số ban đầu thành một heap.

Giai đoạn 2: Sắp xếp dãy số dựa trên heap:

– Bước 1: Đưa phần tử lớn nhất về vị trí đứng ở cuối dãy: r = n; Hoán vị (a1, ar);

– Bước 2: Loại bỏ phần tử lớn nhất ra khỏi heap: r = r-1;

Hiệu chỉnh phần còn lại của dãy từ a1, a2..., ar thành một heap.

– Bước 3: Nếu r>1 (heap còn phần tử): Lặp lại Bước 2

Ngược lại : Dừng

2.2.3. PHƯƠNG PHÁP VUN ĐỐNG (HEAP SORT) (4.1)

4. Cài đặt:

```
void heapify(int arr[], int n, int i) {  // mảng arr, n - số phần tử, i - đỉnh cần vun đống
    int max = i;    // Lưu vị trí đỉnh max ban đầu
    int l = i * 2 + 1;    // Vị trí con trái
    int r = l + 1;    // Vị trí con phải
    if (l<n && arr[l] > arr[max]) {   // Nếu tồn tại con trái lớn hơn cha, gán max = L
        max = l;
    }
    if (r<n && arr[r] > arr[max]) {   // Nếu tồn tại con phải lớn hơn arr[max], gán max = r
        max = r;
    }
    if (max != i) {        // Nếu max != i, tức là cha không phải lớn nhất
        swap(arr[i], arr[max]);   // Đổi chỗ cho phần tử lớn nhất làm cha
        heapify(arr, n, max);    // Đệ quy vun các node phía dưới
    }
}
```

2.2.3. PHƯƠNG PHÁP VUN ĐỐNG (HEAP SORT) (4.2)

4. Cài đặt:

```
// Ham sap xep vun dong
void heapSort(int arr[], int n) {

    // vun dong tu duoi len len de thanh heap
    for (int i = n / 2 - 1; i >= 0; i--)   // i chạy từ n/2 - 1 về 0 sẽ có n/2 đỉnh nhé!
        heapify(arr, n, i);   // Vun từng đỉnh

    // Vòng lặp để thực hiện vun đống và lấy phần tử
    for (int j = n - 1; j > 0; j--) {   // chạy hết j == 1 sẽ dừng
        // mỗi lần j - 1, tương đương với k xét phần tử cuối
        swap(arr[0], arr[j]);  // đổi chỗ phần tử lớn nhất
        heapify(arr, j, 0);    // vun lại đống,
    }
}
```

2.2.4. PHƯƠNG PHÁP PHÂN HOẠCH (QUICK SORT) (1)

1. Ý tưởng:

Áp dụng chiến lược chia để trị:

- Nếu A có không quá 1 phần tử → đã có thứ tự.
- Chọn phần tử chốt (pivot) x
- Chia dãy A thành hai phần:
  - Phần trước chứa Ai sao cho Ai R x
  - Phần sau chứa Aj sao cho x R Aj
- Sắp xếp hai dãy A0,..,Ak-1 và Ak+1,..,An-1 tương tự

2.2.4. PHƯƠNG PHÁP PHÂN HOẠCH (QUICK SORT) (2)

2. Thuật toán:

Input: A={a[0], a[1],…,a[n-1]} chưa có thứ tự R

Output: A={a[0], a[1],…,a[n-1]} đã có thứ tự R

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(n^2)

Trung bình: O(nlogn)

Trường hợp tốt nhất: O(nlogn)

2.2.4. PHƯƠNG PHÁP PHÂN HOẠCH (QUICK SORT) (3)

3. Các bước tiến hành:

Bước 1 : Chọn tùy ý một phần tử a[k] trong dãy là phần tử trục (left ≤ k ≤ right): x = a[k]; i = left; j = right;

Bước 2 : Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ :

- Bước 2a: While (a[i]<x) i++;
- Bước 2b: While (a[j]>x) j--;
- Bước 2c : Nếu (i< j) Swap(a[i],a[j]); // đổi chỗ

Bước 3 : Nếu (i < j): Lặp lại Bước 2.

Ngược lại: Dừng.

2.2.4. PHƯƠNG PHÁP PHÂN HOẠCH (QUICK SORT) (4)

4. Cài đặt:

```
void QuickSort(int a[], int left, int right){
    int i, j, x;
    x = a[(left + right) / 2];
    i = left; j = right;
    while (i <= j)
    {
        while (a[i] < x) i++; // while (a[i] > x) i++;
        while (a[j] > x) j--; // while (a[j] < x ) j--;
        if (i <= j) {
            swap(a[i], a[j]);
            i++; j--;
        }
    }
    if (left < j) QuickSort(a, left, j);
    if (i < right) QuickSort(a, i, right);
}
```

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (1)

1. Ý tưởng:

Áp dụng chiến lược chia để trị:

- Danh sách có 1 phần tử luôn có thứ thự.
- Để sắp xếp danh sách A:

+ Chia A thành hai danh sách A1 và A2

+ Sắp xếp A1 và A2 theo thứ tự R

+ Trộn A1 và A2 theo thứ tự R

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (2)

2. Thuật toán:

Input: A={a[0], a[1],...,a[n-1]} chưa có thứ tự R

Output: A={a[0], a[1],...,a[n-1]} đã có thứ tự R

Độ phức tạp thuật toán:

Trường hợp xấu nhất: O(nlogn)

Trung bình: O(nlogn)

Trường hợp tốt nhất: O(nlogn)

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (3)

3. Các bước tiến hành:

B1: k<-1

B2: Tách A thành hai dãy B và C bằng cách phân phối luân phiên k phần tử cho mỗi dãy

B = a1 ,..,ak ,a2k+1,..,a3k,... C=ak+1,..,a2k,a3k+1,..,a4k,..

B3: Trộn từng cặp dãy con k phần tử của B và C vào A theo thứ tự R

B4: k<-2*k,

B5: Nếu k < n thì qua B2; ngược lại thì kết thúc.

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (4.1)

4. Cài đặt:

```
// Hàm để trộn hai mảng con đã được sắp xếp
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Tạo các mảng tạm thời
    int* L = new int[n1];
    int* R = new int[n2];

    // Sao chép dữ liệu vào các mảng tạm thời L[] và R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
```

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (4.2)

4. Cài đặt:

```
    // Trộn các mảng tạm thời vào mảng arr[]

    i = 0; // Chỉ mục ban đầu của mảng con đầu tiên
    j = 0; // Chỉ mục ban đầu của mảng con thứ hai
    k = l; // Chỉ mục ban đầu của mảng đã được trộn
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {  // Thay đổi điều kiện so sánh thành L[i] >= R[j]
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (4.3)

4. Cài đặt:

```
    // Sao chép các phần tử còn lại của mảng con L[] vào arr[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Sao chép các phần tử còn lại của mảng con R[] vào arr[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
    delete[] L;
    delete[] R;
}
```

2.2.5. PHƯƠNG PHÁP TRỘN (MERGE SORT) (4.4)

4. Cài đặt:

```
// Hàm chính để triển khai thuật toán Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Tương đương với (l+r)/2, nhưng tránh tràn số khi l và r lớn
        int m = (l + (r - l)) / 2;

        // Gọi hàm đệ quy tiếp tục chia đôi từng nửa mảng
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Gộp hai nửa đã được sắp xếp
        merge(arr, l, m, r);
    }
}
```

CHƯƠNG 3: LIST, STACK, QUEUE

3.1 Danh sách liên kết đơn (linked list)

3.2 Ngăn xếp (stack)

3.3 Hàng đợi (queue)

3.1: DANH SÁCH LIÊN KẾT ĐƠN (LINKED LIST)

NỘI DUNG:

3.1.1 Khái niệm

3.1.2 Tổ chức danh sách liên kết đơn

3.1.3 Các thao tác trên danh sách liên kết đơn

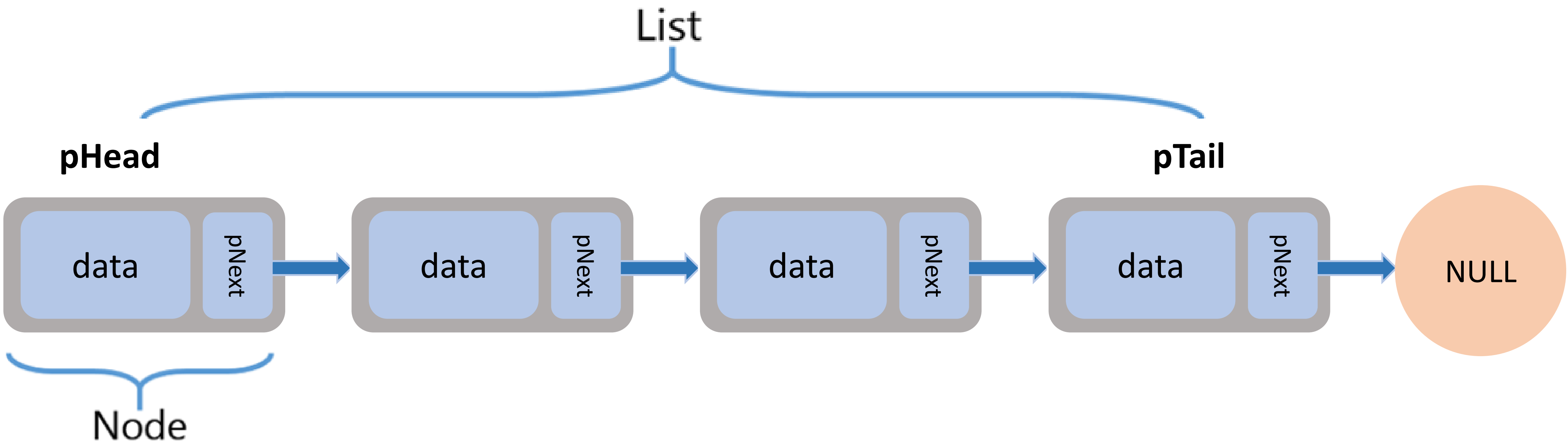
3.1.4 Vận dụng

KHÁI NIỆM

- Danh sách liên kết đơn là tập hợp tuyến tính các phần tử và dữ liệu, mà giữa chúng có dự liên kết với nhau thông qua địa chỉ
- Thành phần trong dslk:

- Thành phần dữ liệu: lưu thông tin
- Thành phần liên kết: lưu địa chỉ của thành phần tiếp theo

TỔ CHỨC DANH SÁCH LIÊN KẾT ĐƠN



CẤU TRÚC DỮ LIỆU CỦA 1 NODE

```
struct Node
{
    int data;
    Node* pNext;
};
```

CẤU TRÚC DỮ LIỆU CỦA DSLK ĐƠN

```
struct List
{
    Node* pHead;
    Node* pTail;
};
```

CÁC THAO TÁC CƠ BẢN TRÊN DSLK ĐƠN

```
void CreateEmptyList(List& list);                        // Tạo một danh sách liên kết đơn rỗng
Node* CreateNode(int x);                                 // Tạo một Node mới
void InsertHead(List& list, Node* p);                    // Thêm một Node vào đầu danh sách
void InsertTail(List& list, Node* p);                    // Thêm một Node vào cuối danh sách
void InsertAfter(Node* prev_node, Node* p);              // Thêm một Node vào địa chỉ cho trước
void Delete(List& list, Node* p);                        // Xóa một Node
void Input(List& list);                                  // Nhập danh sách liên kết đơn
void Output(List& list);                                 // Xuất danh sách liên kết đơn
```

TẠO MỘT DANH SÁCH RỖNG

```
void CreateEmptyList(List& list)
{
    list.pHead = NULL;
    list.pTail = NULL;
}
```

TẠO MỘT NODE MỚI

```
Node* CreateNode(int x)
{
    Node* p = new Node();
    if (p == NULL)
        return NULL;
    p->data = x;
    p->pNext = NULL;
    return p;
}
```

THÊM MỘT NODE VÀO ĐẦU DANH SÁCH

```cpp
void InsertHead(List& list, Node* p)
{
    if (list.pHead == NULL)
        list.pHead = list.pTail = p;
    else
    {
        p->pNext = list.pHead;
        list.pHead = p;
    }
}
```

THÊM MỘT NODE VÀO CUỐI DANH SÁCH

```cpp
void InsertTail(List& list, Node* p)
{
    if (list.pHead == NULL)
        list.pHead = list.pTail = p;
    else
    {
        list.pTail->pNext = p;
        list.pTail = p;
    }
}
```

THÊM MỘT NODE VÀO ĐỊA CHỈ CHO TRƯỚC

```
void InsertAfter(Node* prev_node, Node* p)
{
    if (prev_node == NULL)
        return;
    p->pNext = prev_node->pNext;
    prev_node->pNext = p;
}
```

XÓA MỘT NODE

```cpp
void Delete(List& list, Node* p)
{
    if (list.pHead == NULL)
        return;
    else if (p == list.pHead)
        list.pHead = list.pHead->pNext;
    else
    {
        for (Node* q = list.pHead->pNext; q != NULL; q = q->pNext)
        {
            if (q->pNext == p)
            {
                if (q->pNext == list.pTail)
                {
                    list.pTail = q;
                    list.pTail->pNext = NULL;
                }
                else
                    q->pNext = p->pNext;
                break;
            }
        }
    }
    delete p;
}
```

NHẬP DSLK ĐƠN

```
void Input(List& list)
{
    int x, n;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> x;
        Node* p = CreateNode(x);
        InsertTail(list, p);
    }
}
```

XUẤT DSLK ĐƠN

```
void Output(List& list)
{
    Node* p = list.pHead;
    while (p != NULL)
    {
        cout << p->data << " ";
        p = p->pNext;
    }
}
```

VẬN DỤNG

(DSLK Đơn) Người ta muốn lưu trữ danh sách hàng hóa tại một cửa hàng X với các thông tin chính yếu nhằm hỗ trợ nhanh trong tra cứu, với các thông tin: Tên mặt hàng (chuỗi); Giá mặt hàng (số nguyên); Số lượng trong kho (số nguyên). Hãy thực hiện:

a. Định nghĩa cấu trúc dữ liệu lưu danh sách các mặt hàng theo thông tin mô tả ở trên, sử dụng cấu trúc danh sách liên kết.

b. Viết hàm nhập vào danh sách n mặt hàng sử dụng cấu trúc dữ liệu ở câu a, biết rằng khi nhập lần lượt từng mặt hàng sẽ thêm vào cuối danh sách.

c. Viết hàm nhập vào số nguyên dương x, hiển thị lên màn hình danh sách mặt hàng có số lượng ít hơn x

VẬN DỤNG

INPUT	OUPUT
5 APPLE 4000 100 MANGO 15000 10 BANANA 6000 50 ORANGE 10000 20 WATERMELON 30000 5 30	Danh sach hang hoa du dieu kien: ----------------------- Name: APPLE Price: 4000 Quantity: 100 ----------------------- Name: BANANA Price: 6000 Quantity: 50

3.2: NGĂN XẾP (STACK)

NỘI DUNG:

3.2.1 Khái niệm

3.2.2 Các thao tác trên ngăn xếp

3.2.3 Các cách cài đặt của ngăn xếp

3.2.4 Vận dụng

KHÁI NIỆM

Stack (ngăn xếp) là một cấu trúc dữ liệu trong lập trình, hoạt động theo nguyên tắc "vào sau, ra trước" (LIFO - Last-In, First-Out). Nghĩa là phần tử cuối cùng được thêm vào stack sẽ là phần tử đầu tiên được lấy ra khỏi stack.

CÁC THAO TÁC TRÊN NGĂN XẾP

•push(x): Thêm phần tử x vào đỉnh của stack.
•pop(): Xóa phần tử đầu tiên (đỉnh) khỏi stack.
•top(): Truy cập và trả về giá trị của phần tử đầu tiên (đỉnh) trong stack.
•empty(): Kiểm tra xem stack có rỗng hay không.
•size(): Trả về số lượng phần tử trong stack.

CÁC CÁCH CÀI ĐẶT CỦA NGĂN XẾP

- Dùng mảng 1 chiều
- Dùng danh sách liên kết đơn
- Dùng thư viện

Chú ý: thêm và hủy cùng 1 phía

CẤU TRÚC DỮ LIỆU STACK

```
struct Stack
{
    int top;
    int data[100];
};
```

KHỞI TẠO STACK RỖNG

```
void CreateStack(Stack& s)
{
    s.top = -1;
}
```

CÁC THAO TÁC CƠ BẢN TRÊN STACK

```
bool IsEmpty(Stack& s);                 // Kiểm tra stack rỗng
bool IsFull(Stack& s);                  // Kiểm tra stack đầy
void push(Stack& s, int x);             // Thêm một phần tử vào đầu stack
void pop(Stack& s);                     // Xóa phần tử đầu của stack
int top(Stack& s);                      // Lấy giá trị của phần tử ở đỉnh stack
int size(Stack& s);                     // Trả về số lượng phần tử trong stack
```

KIỂM TRA STACK RỖNG

```
bool IsEmpty(Stack& s)
{
    if (s.top == -1)
        return true;
    return false;
}
```

KIỂM TRA STACK ĐẦY

```
bool IsFull(Stack& s)
{
    if (s.top >= 100)
        return true;
    return false;
}
```

THÊM MỘT PHẦN TỬ VÀO ĐẦU STACK

```
void push(Stack& s, int x)
{
    if (IsFull(s))
        return;
    s.top++;
    s.data[s.top] = x;
}
```

XÓA MỘT PHẦN TỬ ĐẦU CỦA STACK

```
void pop(Stack& s)
{
    if (IsEmpty(s))
        return;
    s.top--;
}
```

LẤY GIÁ TRỊ CỦA PHẦN TỬ Ở ĐỈNH STACK

```
int top(Stack& s)
{
    return s.data[s.top];
}
```

TRẢ VỀ SỐ LƯỢNG PHẦN TỬ TRONG STACK

```
int size(Stack& s)
{
    return s.top + 1;
}
```

GIỚI THIỆU VỀ CÁCH DÙNG THƯ VIỆN STACK

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> myStack;

    myStack.push(1);                        // Thêm 1 phần tử vào đỉnh
    myStack.push(2);
    myStack.push(3);

    while (!myStack.empty()) {              // Kiểm tra r
        std::cout << myStack.top() << " ";  // In giá trị phần tử đỉnh
        myStack.pop();                      // Xóa phần tử đỉnh
    }

    return 0;
}
```

3.3: HÀNG ĐỢI (QUEUE)

NỘI DUNG:

3.3.1 Khái niệm

3.3.2 Các thao tác trên hang đợi

3.3.3 Các cách cài đặt của hàng đợi

3.3.4 Vận dụng

KHÁI NIỆM

Hàng đợi (Queue) là một cấu trúc dữ liệu trong lập trình, có thể được hiểu như một danh sách các phần tử tuân theo nguyên tắc "First-In-First-Out" (FIFO), tức là phần tử đầu tiên được thêm vào hàng đợi sẽ là phần tử đầu tiên được lấy ra khỏi hàng đợi.

CÁC THAO TÁC TRÊN HÀNG ĐỢI

- Enqueue: Thêm một phần tử vào hàng đợi, đặt phần tử mới vào phía sau (rear) của hàng đợi.

- Dequeue: Lấy ra phần tử từ hàng đợi, phần tử đầu tiên (phía trước - front) sẽ được lấy ra khỏi hàng đợi.

- Front: Truy cập vào phần tử đầu tiên trong hàng đợi mà không xóa nó.

- IsEmpty: Kiểm tra xem hàng đợi có rỗng hay không.

- IsFull: Kiểm tra xem hang đợi có đầy hay không.

- Size: Lấy số lượng phần tử hiện có trong hàng đợi.

CÁC CÁCH CÀI ĐẶT CỦA HÀNG ĐỢI

- Dùng mảng 1 chiều
- Dùng danh sách liên kết đơn
- Dùng thư viện

CẤU TRÚC DỮ LIỆU QUEUE

```
struct Queue
{
    int n;
    int data[100];
};
```

KHỞI TẠO QUEUE RỖNG

```
void CreateQueue(Queue& q)
{
    q.n = 0;
}
```

CÁC THAO TÁC CƠ BẢN TRÊN QUEUE

```
bool IsEmpty(Queue& q);                    // Kiểm tra Queue có rỗng không
bool IsFull(Queue& q);                     // Kiểm tra Queue có đầy không
void enqueue(Queue& q, int x);             // Thêm một phần tử vào Queue
void dequeue(Queue& q);                    // Xóa một phần tử của Queue
int front(Queue& q);                       // Truy cập vào phần tử đầu của Queue
int size(Queue& q);                        // Trả về số lượng của Queue
```

KIỂM TRA QUEUE RỖNG

```
bool IsEmpty(Queue& q)
{
    if (q.n == 0)
        return true;
    return false;
}
```

KIỂM TRA QUEUE RỖNG

```
bool IsFull(Queue& q)
{
    if (q.n >= 100)
        return true;
    return false;
}
```

THÊM MỘT PHẦN TỬ VÀO QUEUE

```
void enqueue(Queue& q, int x)
{
    if (IsFull(q))
        return;
    q.data[q.n] = x;
    q.n++;
}
```

XÓA MỘT PHẦN TỬ CỦA QUEUE

```
void dequeue(Queue& q)
{
    if (IsEmpty(q))
        return;
    for (int i = 0; i < q.n - 1; i++)
        q.data[i] = q.data[i + 1];
    q.n--;
}
```

LẤY GIÁ TRỊ ĐẦU TIÊN CỦA QUEUE

```
int front(Queue& q)
{
    return q.data[0];
}
```

TRẢ VỀ SỐ LƯỢNG PHẦN TỬ TRONG QUEUE

```
int size(Queue& q)
{
    return q.n;
}
```

GIỚI THIỆU VỀ CÁCH DÙNG THƯ VIỆN QUEUE

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> myQueue;
    myQueue.push(1);                              // Thêm 1 phần tử
    myQueue.push(2);
    myQueue.push(3);
    myQueue.push(4);

    while (!myQueue.empty())                     // Kiểm tra rỗng
    {
        std::cout << myQueue.front() << " ";     // In giá trị đầu tiên
        myQueue.pop();                           // Xóa phần tử đầu tiên
    }

    return 0;
}
```

VẬN DỤNG

(Stack, Queue) Cho chuỗi S chỉ gồm các số nguyên dương chỉ gồm một chữ số và các dấu “+”, “-”, “*”, “:”.

Hãy tính giá trị của biểu thức được biểu diễn bởi chuỗi S đó (biết rằng trong S không chứa khoảng trắng).

INPUT	OUTPUT
1+2+3	6
1+2*3+2	9
1+2*3:4*5-2	6.5

CHƯƠNG 4: TREE

4.1 Cây và Cây nhị phân (Tree and Binary Tree)

4.2 Cây nhị phân tìm kiếm (Binary Search Tree)

4.3 B-Tree

4.1: CÂY VÀ CÂY NHỊ PHÂN (TREE AND BINARY TREE)

NỘI DUNG:

4.1.1 Định nghĩa

4.1.2 Một số khái niệm

4.1.3 Cây nhị phân

ĐỊNH NGHĨA

Cây là một tập hợp T các phần tử (gọi là nút của cây), trong đó có một nút đặc biệt gọi là nút gốc, các nút còn lại được chia thành những tập rời nhau T_1 , T_2 , ...,T_n theo quan hệ phân cấp, trong đó T_i cũng là 1 cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta gọi là quan hệ cha – con.

CÁC KHÁI NIỆM CƠ BẢN

1. Nút (Node): Mỗi phần tử trong cây được gọi là một nút. Mỗi nút có một giá trị và có thể có một hoặc nhiều nút con.

2. Nút gốc (Root Node): Là nút đỉnh cao nhất trong cây, không có nút cha. Cây chỉ có một nút gốc.

3. Nút con (Child Node): Là các nút kết nối trực tiếp với một nút cha. Một nút có thể có nhiều nút con.

4. Nút cha (Parent Node): Là nút kết nối trực tiếp với một hoặc nhiều nút con.

5. Nút lá (Leaf Node): Là các nút không có nút con, nằm ở mức cuối cùng của cây.

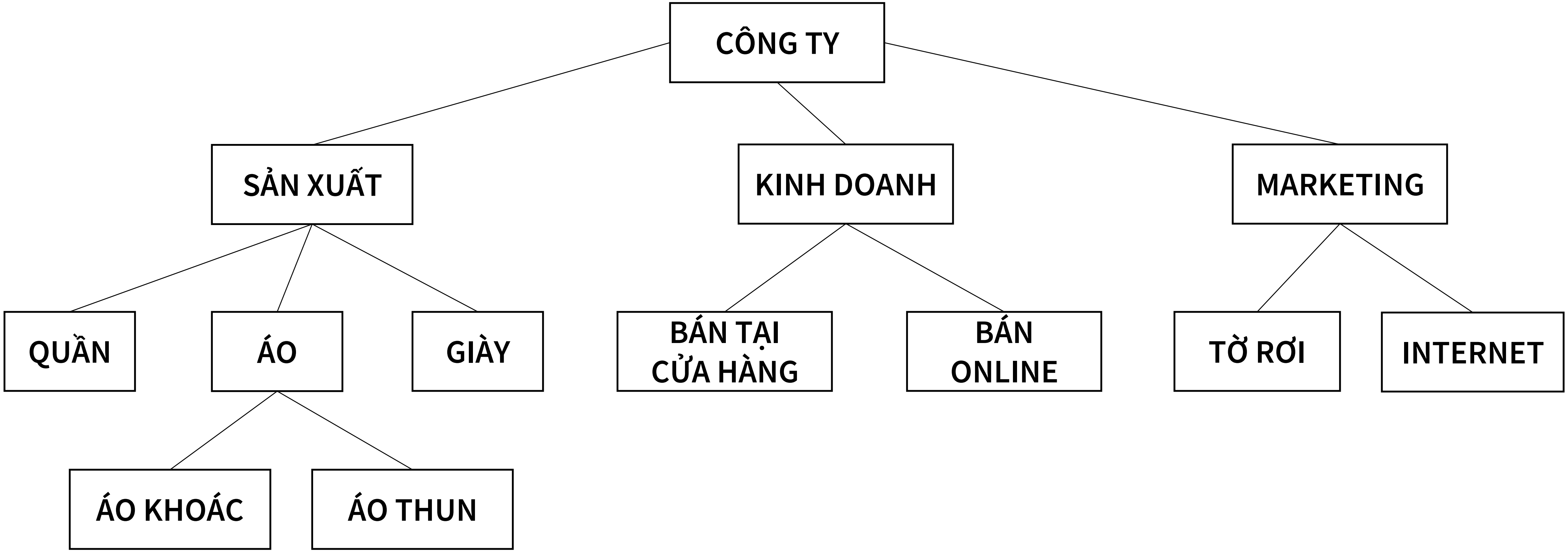
6. Đường đi (Path): Đường đi từ node n1 tới nk được định nghĩa là: Một tập các node n1, n2, ..., nk sao cho ni là cha của ni+i ($1 \leq i < k$)

7. Bậc của một nút (Degree of Node): Bậc của một nút là số cây con của nút đó

8. Bậc của một cây (Degree of tree): Là bậc lớn nhất của các nút trong cây

9. Độ sâu của cây (The depth of a tree): bằng chiều đường đi từ nút gốc tới nút lá sâu nhất.

VÍ DỤ


CÔNG TY
SẢN XUẤT
KINH DOANH
MARKETING
QUẦN
ÁO
GIÀY
BÁN TẠI CỬA HÀNG
BÁN ONLINE
TỜ RƠI
INTERNET
ÁO KHOÁC
ÁO THUN

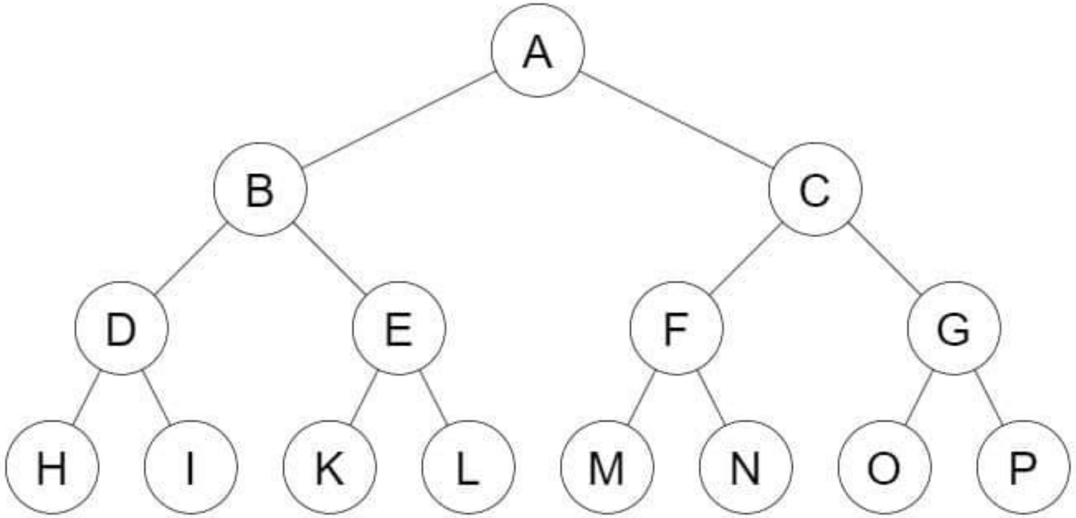
CÂY NHỊ PHÂN

Mỗi node có tối đa 2 cây con
Một số tính chất:

- Số nút nằm ở mức i tối đa là 2^i.
- Số nút lá ≤ 2^h, với h là chiều cao của cây.
- Chiều cao của cây h ≥ $\log_2(N)$ với N = số nút trong cây
- Số nút trong cây ≤ $2^{h+1} - 1$

Các loại cây nhị phân:

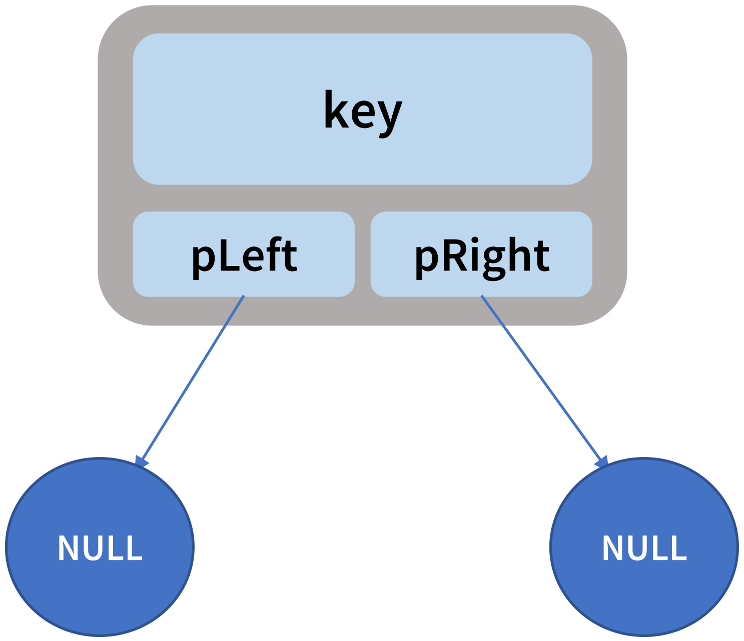
- Cây nhị phân hoàn chỉnh
- Cây nhị phân đầy đủ



CẤU TRÚC DỮ LIỆU CÂY NHỊ PHÂN

```
struct TNode
{
    int key;
    TNode* pLeft;
    TNode* pRight;
};

typedef TNode* TREE;
```



DUYỆT CÂY NHỊ PHÂN

Có 3 trình tự thăm gốc: (3 cách cơ bản để duyệt cây)
+ Duyệt tiền thứ tự (preorder Traversal) : gốc – trái – phải.

+ Duyệt trung thứ tự (inorder traversal): trái – gốc – phải.

+ Duyệt hậu thứ tự (postorder traversal): trái – phải – gốc.

Độ phức tạp O ($\log_2(h)$)
+ Trong đó h là chiều cao cây

DUYỆT TIỀN THỨ TỰ

```
void preorder(TREE Root) {
    if (Root ≠ NULL) {
        <Xử lý Root>; //Xử lý tương ứng theo nhu cầu
        preorder(Root→pLeft);
        preorder(Root→pRight);
    }
}
```

DUYỆT TRUNG THỨ TỰ

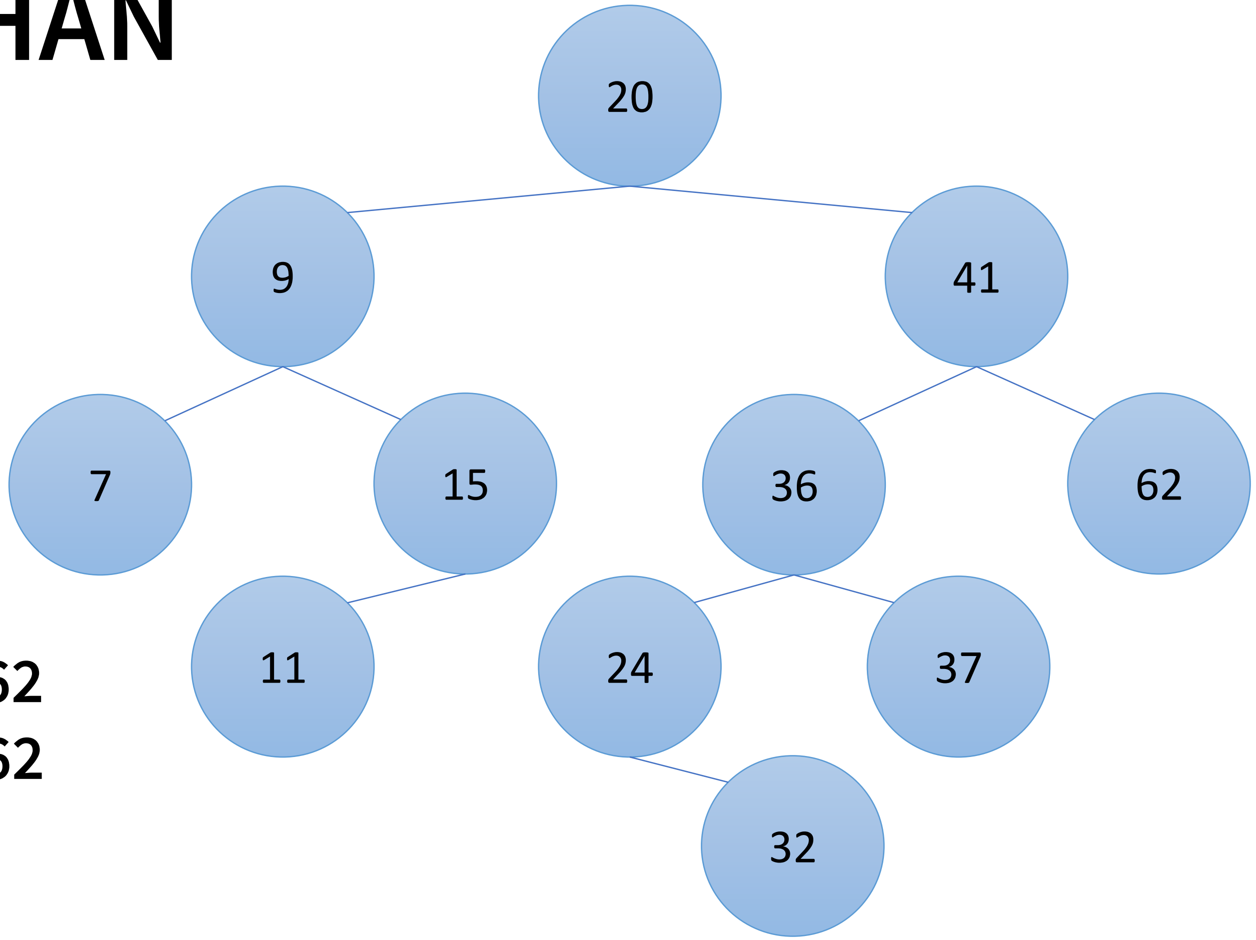
```
void inorder(TREE Root) {
    if (Root != NULL) {
        inorder(Root->pLeft);
        <Xử lý Root>; //Xử lý tương ứng theo nhu cầu
        inorder(Root->pRight);
    }
}
```

DUYỆT HẬU THỨ TỰ

```
void postorder(TREE Root) {
    if (Root != NULL) {
        postorder(Root->pLeft);
        postorder(Root->pRight);
        <Xử lý Root>; //Xử lý tương ứng theo nhu cầu
    }
}
```

DUYỆT CÂY NHỊ PHÂN

Ví Dụ:



Kết quả:
NLR: 20 9 7 15 11 41 36 24 32 37 62
LNR: 7 9 11 15 20 24 32 36 37 41 62
LRN: 7 11 15 32 24 37 36 62 41 6

4.2: CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE)

NỘI DUNG:

4.2.1 Định nghĩa

4.2.2 Một thao tác trên cây nhị phân tìm kiếm

4.2.3 Vận dụng

ĐỊNH NGHĨA

Cây nhị phân:
Bảo đảm nguyên tắc bố trí khoá tại mỗi node:

- Các node trong cây trái nhỏ hơn node hiện hành
- Các node trong cây phải lớn hơn node hiện hành

CÁC THAO TÁC TRÊN CÂY NHỊ PHÂN

- Tạo 1 cây rỗng
- Tạo 1 nút có giá trị bằng x
- Thêm 1 nút vào cây nhị phân tìm kiếm
- Tìm 1 nút có key bằng x trên cây
- Xóa 1 nút có key bằng x trên cây

TẠO 1 CÂY RỖNG

```cpp
// Tạo cây rỗng
void CreateEmptyTree (TREE& T)
{
    T = NULL;
}
```

TẠO 1 NODE MỚI

```cpp
// Tạo một nút mới với giá trị key
TNODE* CreateNode(int key)
{
    TNODE* Node = new TNODE();
    if (Node == NULL)
        return NULL;
    Node->key = key;
    Node->pLeft = NULL;
    Node->pRight = NULL;
    return Node;
}
```

THÊM 1 NODE VÀO CÂY

```
// Thêm một nút vào cây
void InsertNode(TREE& T, int key)
{
    if (T == NULL)
    {
        T = CreateNode(key);
        return;
    }
    if (key == T->key)
        return;
    else if (key < T->key)
        InsertNode(T->pLeft, key);
    else
        InsertNode(T->pRight, key);
}
```

TÌM 1 NODE CÓ KEY BẰNG X

```
// Tìm 1 nút có key bằng x trên cây
TNODE* searchNode(TREE T, int x)
{
    if (T != NULL)
    {
        if (T->key == x)
            return T;
        if (T->key > x)
            return searchNode(T->pLeft, x);
        return searchNode(T->pRight, x);
    }
    return NULL;
}
```

CÁC THAO TÁC HỦY TRÊN CÂY NHỊ PHÂN

Hủy 1 nút có khóa bằng x trên cây

Hủy 1 phần tử trên cây phải đảm bảo điều kiện ràng buộc của cây nhị phân tìm kiếm

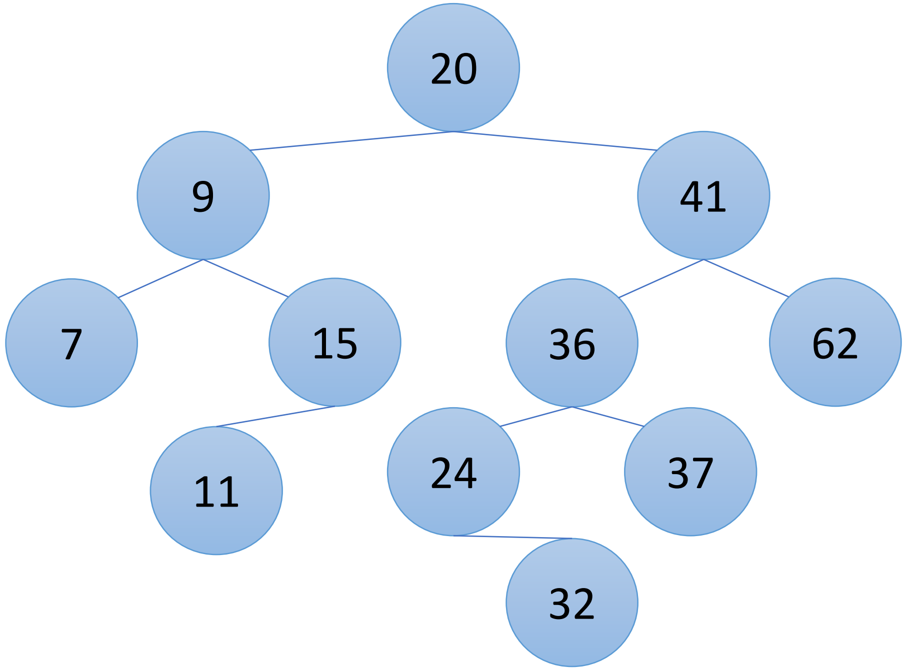
Có 3 trường hợp khi hủy 1 nút trên cây:

- TH1: X là nút lá
- TH2: X có 1 cây con (trái hoặc phải)
- TH3: X có 2 cây con

TRƯỜNG HỢP 1: X LÀ NODE LÁ

- Ta xóa nút lá mà không ảnh hưởng đến các nút khác trên cây

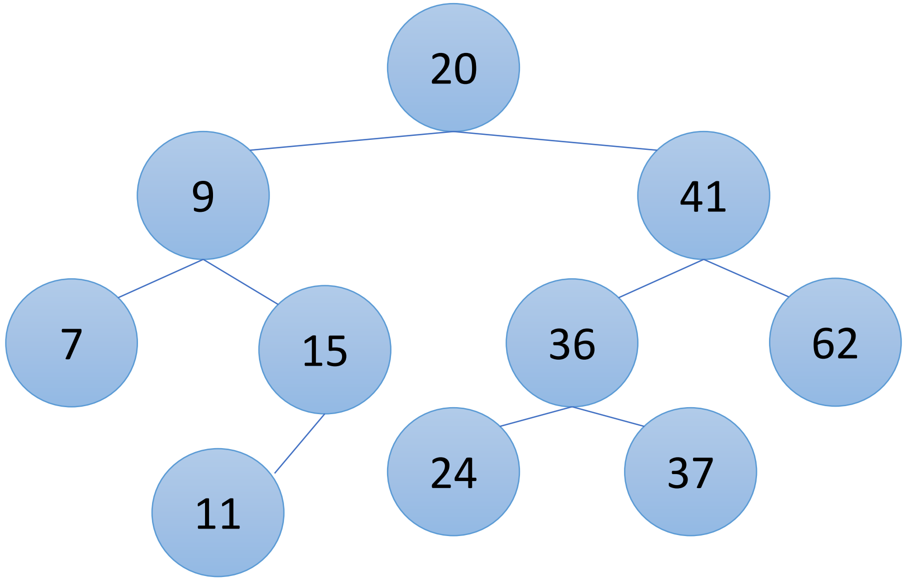
Hủy x = 32



TRƯỜNG HỢP 2: X CÓ MỘT NODE CON

- Ta xóa nút lá mà không ảnh hưởng đến các nút khác trên cây

Hủy x = 15



TRƯỜNG HỢP 3: X CÓ HAI CÂY CON (XÓA GIÁN TIẾP)

- Thay vì hủy X ta tìm phần tử thế mạng Y. Nút Y có tối đa 1 cây con.
- Thông tin lưu tại nút Y sẽ được chuyển lên lưu tại X.
- Ta tiến hành xóa hủy nút Y (xóa Y giống 2 trường hợp đầu).
- Cách tìm nút thế mạng Y cho X: có 2 cách:

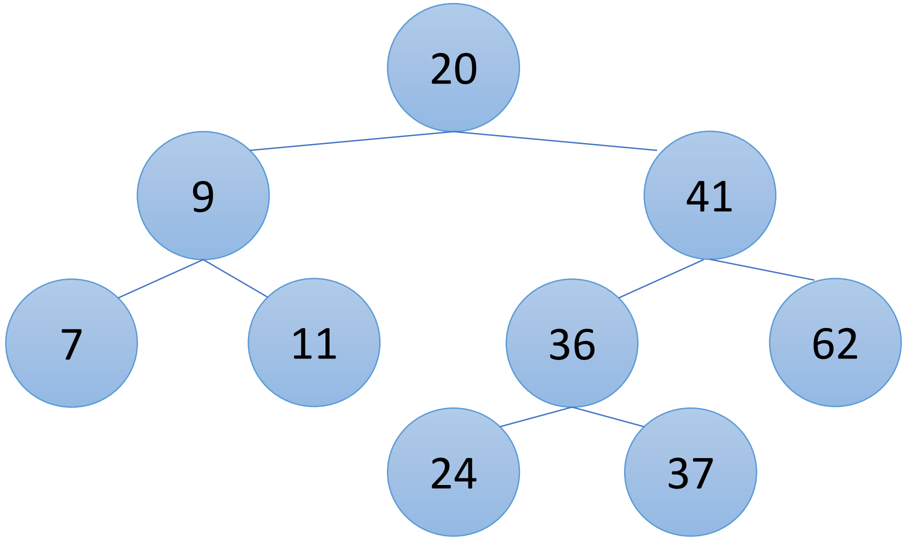
  C1: Nút Y là nút có khóa nhỏ nhất (trái nhất) bên cây con phải X.

  C2: Nút Y là nút có khóa lớn nhất (phải nhất) bên cây con trái của X.

TRƯỜNG HỢP 1: X LÀ NODE LÁ

- Trước khi xóa X ta móc nối cha của X với con duy nhất của X

Hủy x = 41



XÓA 1 NODE CÓ KEY BẰNG X TRÊN CÂY

- Xóa phần tử có 2 cây con

```
// Tìm phần tử thay thế bên cây con phải
void NodeReplace(TREE& p, TREE& T)
{
    if (T->pLeft != NULL)
        NodeReplace(p, T->pLeft);
    else {
        p->key = T->key;
        p = T;
        T = T->pRight;
    }
}
```

```
// Xóa 1 nút có ke bằng x trên cây
void DeleteNode(TREE& T, int x)
{
    if (T != NULL) {
        if (T->key < x)
            DeleteNode(T->pRight, x);
        else {
            if (T->key > x)
                DeleteNode(T->pLeft, x);
            else {
                TNODE* p = T;
                if (T->pLeft == NULL)
                    T = T->pRight;
                else {
                    if (T->pRight == NULL)
                        T = T->pLeft;
                    else
                        NodeReplace(p, T->pRight);
                    delete p;
                }
            }
        }
    }
}
```

4.3: B-TREE

Nội dung:
4.3.1: Định nghĩa
4.3.2: Tìm kiếm
4.3.3: Thêm
4.3.4: Xóa

ĐỊNH NGHĨA

Một B-Tree bậc M là cây M-Phân tìm kiếm có các tính chất sau:

- Mỗi node (khác node gốc và lá) chứa ít nhất [M/2] cây con và nhiều nhất M cây con (làm tròn lên)
- Mỗi node (trừ node gốc) có từ [M/2] – 1 khóa đến M - 1 khóa, node gốc từ 1 đến M - 1
- Tại mỗi khóa X bất kỳ, các khóa ở nhánh trái < x < các khóa ở nhánh phải
- Các node là nằm cùng mức

TÌM KIẾM

$$C_1 \; K_1 \; C_2 \; K_2 \ldots C_n \; K_n \; C_{n+1}$$

Các trường hợp xảy ra khi tìm 1 node X. Nếu X không tìm thấy sẽ có 3 trường hợp sau xảy ra:

o Ki < X < Ki+1 : Tiếp tục tìm kiếm trên cây con Ci+1

o Km < X : Tiếp tục tìm kiếm trên Cn+1

o X < K1: Tiếp tục tìm kiếm trên C1

• Quá trình này tiếp tục cho đến khi node được tìm thấy. Nếu đã đi đến node lá mà vẫn không tìm thấy khoá, việc tìm kiếm là thất bại.

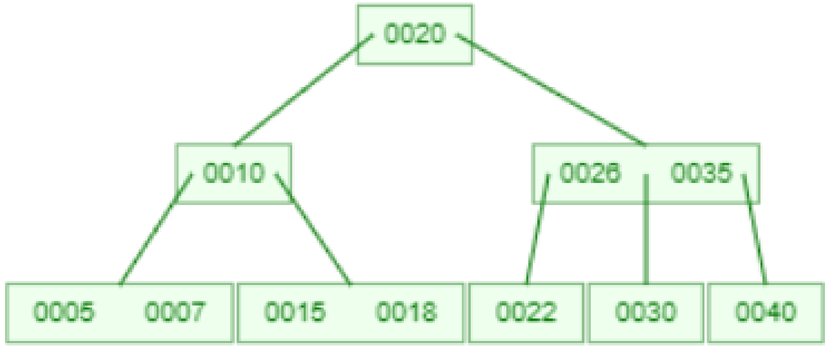
THÊM

Khóa mới sẽ được thêm vào node lá:

- Nếu chưa đầy -> Thêm được
- Nếu đầy -> tách node:
  - Khóa giữa node được lan truyền ngược lên node cha (Trong những trường hợp đặc biệt lan truyền đến tận gốc của B-Tree)
  - Phần còn lại chia thành 2 node cạnh nhau trong cùng 1 mức

VÍ DỤ:

- Tạo B-Tree bậc 3 từ dãy các khóa sau : 20,40,10,30,15,35,7,26,18,22,5



Link tham khảo: https://bom.so/QFULnq

XÓA

- Khóa cần xóa nằm trên node lá -> Xóa bình thường
- Khóa cần xóa không trên node lá
  - Tìm phần tử thay thế: trái nhất ở cây con bên phải hoặc phải nhất của cây con bên trái

CHƯƠNG 5: BẢNG BĂM

Ý tưởng:

- Dùng cấu trúc mảng để truy xuất ngẫu nhiên.
- Ánh xạ giá trị cần tìm vào tập chỉ số của mảng

→ Tìm bằng cách truy xuất ngẫu nhiên phần tử

của mảng.

5.1. Băm

5.2. Xử lý đụng độ

5.3. Băm lại

5.1. BĂM (1)

- **Khái niệm bảng băm:** Bảng băm là một CTDL trong đó mỗi phần tử là một cặp (khoá, giá trị)
- **Đặc điểm:**

+ Có kích thước m xác định.

+ Cho phép truy xuất ngẫu nhiên từng phần tử theo giá trị khóa.

+ Độ phức tạp thời gian có thể đạt O(1) khi truy xuất phần tử.

-> Một bảng băm tốt cần phải có hàm băm tốt. Quá trình ánh xạ khóa vào bảng băm được thực hiện thông qua hàm băm

5.1. BĂM (2)

- **Hàm băm là** một ánh xạ biến đổi khóa thành chỉ số của mảng
- **Hàm băm đủ tốt:**

+ Tính toán nhanh (không phải là thuật toán)
+ Các khóa được phân bố đều trong bảng
+ Ít xảy ra đụng độ
+ Giải quyết vấn đề băm với các khóa không là số nguyên

- **Các dạng hàm băm**

+ Hàm băm dùng phép chia: h(k) = k % m
+ Hàm băm dùng phép nhân: h(k) = [m * (k*A % 1)]
+ Hàm băm phổ quát: h(k) ={ha,b(k)=((a*k+b) % p) % m}

5.2. XỬ LÝ ĐỤNG ĐỘ (1)

- **Khái niệm sự đụng độ:** Hiện tượng các khóa khác nhau nhưng băm cùng địa chỉ như nhau
- **Hai phương pháp:**

+ Phương pháp nối kết:

Nối kết trực tiếp

Nối kết hợp nhất

+ Phương pháp địa chỉ mở:

Dò tuyến tính.

Dò bậc 2.

Băm kép.

5.2. XỬ LÝ ĐỤNG ĐỘ (2)

Phương pháp		Hàm băm	Giải quyết xung đột
Nối kết	Trực tiếp – Direct Chaining	H(key)=key%M	Gom thành 1 DSLK
	Hợp nhất – Coalesced Chaining	H(key)=key%M	Nút trống phía cuối mảng
Địa chỉ mở	Dò tuyến tính – Linear Probing	H(key)=key%M	H’(key)=(H(key)+i)%M
	Dò bậc 2 – Quadratic Probing	H(key)=key%M	H’(key)=(H(key)+i^2)%M
	Băm kép – Double Hashing	H_1(key)=key%M H_2(key)=(M-2)-key%(M-2)	H’(key)=(H_1(key)+i*H_1(key))%M
i = 1, 2, …, M-1 (M là kích thước bảng băm)			

5.3 BĂM LẠI

- Băm lại (rehashing) cần được thực hiện khi:

+ Không thể thực hiện một thao tác thêm phần tử mới bất kỳ.

+ Bảng băm có hệ số tải đạt một giá trị xác định ($\lambda \geq 0.5$) làm độ phức tạp thời gian của thao tác tìm kiếm tăng

- Các bước thực hiện khi băm lại:

+ Tăng kích thước bảng băm lên m’ với m’ là một số nguyên tố và m’ $\approx 2 * m$.

+ Cập nhật hàm băm.

+ Băm lại các phần tử có trong bảng băm cũ và thêm vào bảng băm mới

CHƯƠNG 6: ĐỒ THỊ

6.1: CÁC KHÁI NIỆM TRÊN ĐỒ THỊ

NỘI DUNG:

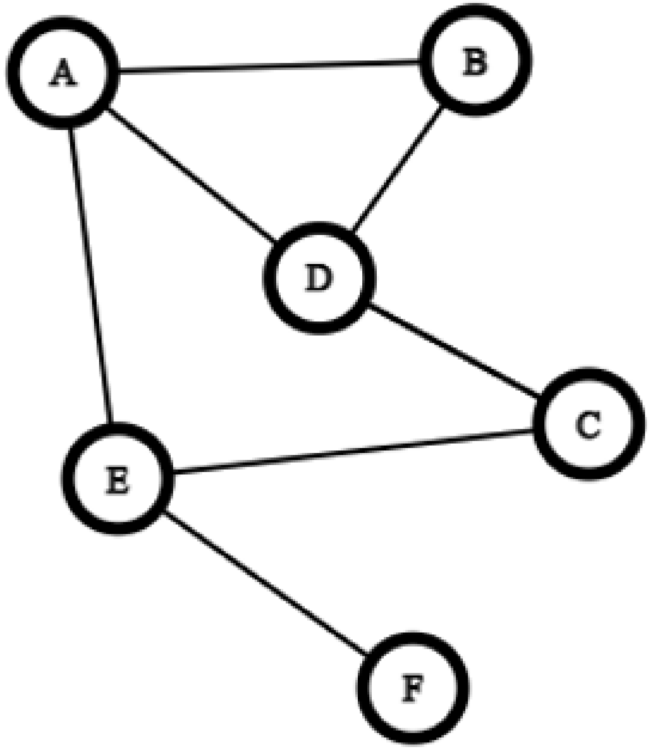
6.1.1 Định nghĩa

6.1.2 Các loại đồ thị

6.1.3 Khái niệm đường đi chu trình, liên thông

ĐỊNH NGHĨA

Đồ thị (Graph) bao gồm:
- Tập đỉnh (Vertices)
- Tập cạnh (Edges)



ĐỒ THỊ VÔ HƯỚNG

Một đồ thị vô hướng G = (V, E) được định nghĩa bởi:

– Tập hợp V được gọi là tập các đỉnh của đồ thị;

– Tập hợp E là tập các cạnh của đồ thị;

– Mỗi cạnh e ∈ E được liên kết

với một cặp đỉnh {i,j} ∈ V^2,không phân biệt thứ tự

ĐỒ THỊ CÓ HƯỚNG

Một đồ thị có hướng G = (V,U) được định nghĩa bởi:

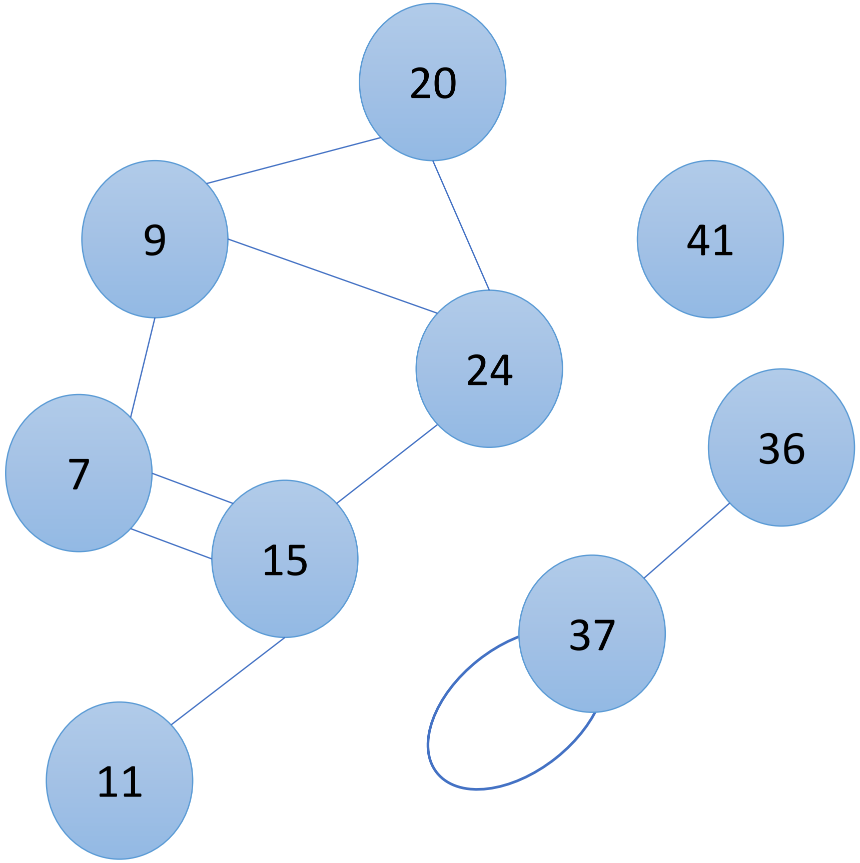
– Tập hợp V được gọi là tập các đỉnh của đồ thị;

– Tập hợp U là tập các cạnh của đồ thị;

– Mỗi cạnh u ∈ U được liên kết với một cặp đỉnh (i,j) ∈ V^2

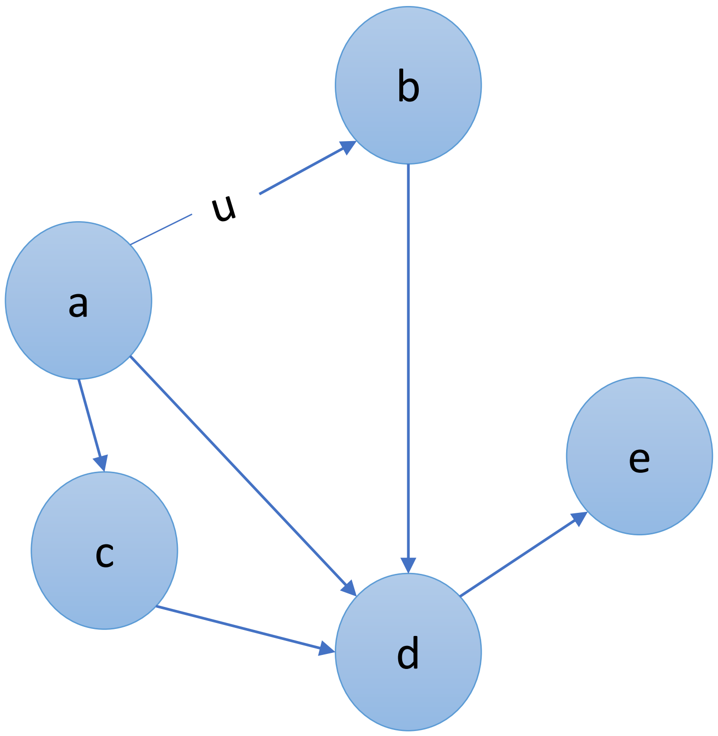
CÁC KHÁI NIỆM ĐỒ THỊ VÔ HƯỚNG

- Đỉnh kề: Hai đỉnh được nối với nhau bằng 1 cạnh kề
- Bậc của đỉnh: Bậc của đỉnh trong đồ thị là số các cạnh kề với đỉnh đó ,mỗi khuyên được tính hai lần.
- Khuyên (loop): Cạnh có hai đầu trùng nhau (cùng một đỉnh).
- Một đỉnh cô lập (isolated vertex) hoặc đỉnh độc lập là đỉnh không liên thuộc với một cạnh nào, cũng có nghĩa là đỉnh có bậc 0.
- Đỉnh có bậc 1 là một đỉnh treo hay lá (leaf vertex, pendant vertex).
- Cạnh song song (parallel edge): tồn tại nhiều hơn 1 cạnh giữa 2 đỉnh.



ĐỒ THỊ CÓ HƯỚNG

- Cạnh u kề với đỉnh a và đỉnh b (hay đỉnh a và đỉnh b kề với cạnh u); có thể viết tắt u = (a,b). Cạnh u đi ra khỏi đỉnh a và đi vào đỉnh b
- Đỉnh b được gọi là đỉnh kề (adjacency vertex) của đỉnh a. Xét đồ thị có hướng G
- Nửa bậc ngoài của đỉnh x là số các cạnh đi ra khỏi đỉnh x, ký hiệu: $d^+(x)$
- Nửa bậc trong của đỉnh x là số các cạnh đi vào đỉnh x, ký hiệu: $d^-(x)$
- Bậc của đỉnh x: $\mathrm{d}(\mathrm{x}) = d^+(x) + d^-(x)$

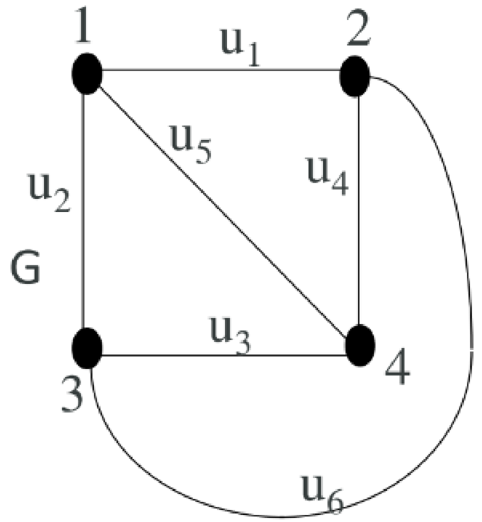


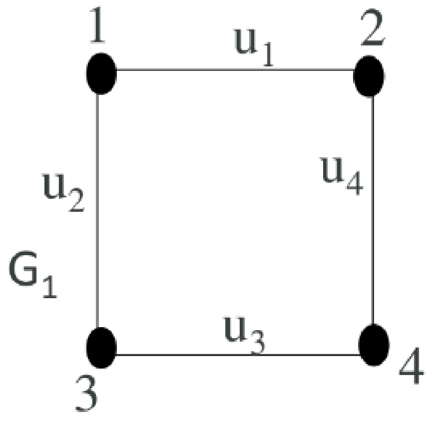
ĐỒ THỊ CON

Xét hai đồ thị G = (X,U) và G1 = (X1,U1). G1 được gọi là đồ thị con của G và ký hiệu G1 ∈ G nếu:

– X1 ∈ X; U1 ∈ U

– u = (i,j) ∈ U của G, nếu u ∈ U1 thì i,j ∈ X1





ĐƯỜNG ĐI, CHU TRÌNH

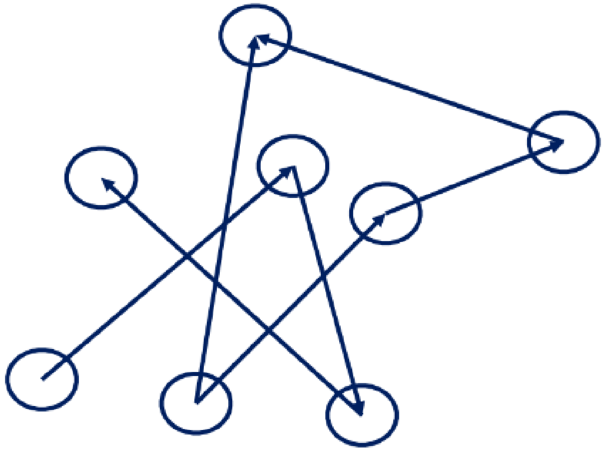
•Một đường đi trong G = (V,U) là một đồ thị con C = (V, E) của G với:

- V = $\{x_1, x_2 \ldots\ , x_M\}$
- E = $\{u_1, u_2\ \ \ldots, u_{M-1}\}$ với $\{u_1\ = x_1x_2,\ \ u_{21}\ = x_2x_3, \ldots, u_{M-1}\ \ = x_{M-1}x_M\}$; liên kết x_jx_{j+1} không phân biệt thứ tự.

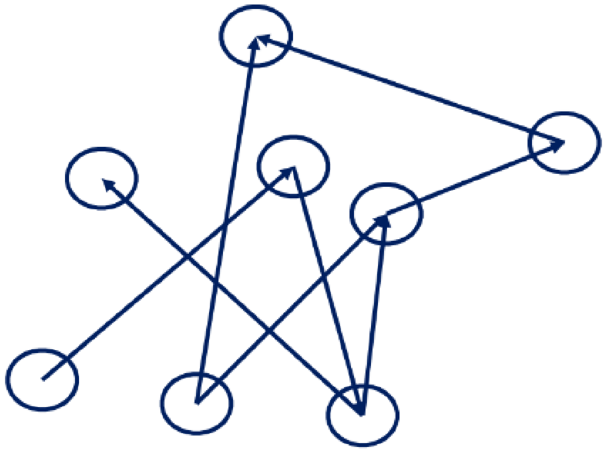
•Khi đó, x_1 và x_M được nối với nhau bằng đường đi C với: C. x_1 là đỉnh đầu và C. x_M là đỉnh cuối.

LIÊN THÔNG

- G gồm 2 thành phần liên thông, H là đồ thị liên thông.



G



H

6.2: Biểu diễn đồ thị trên máy tính

NỘI DUNG:

6.2.1 Ma trận kề

6.2.2 Danh sách cạnh

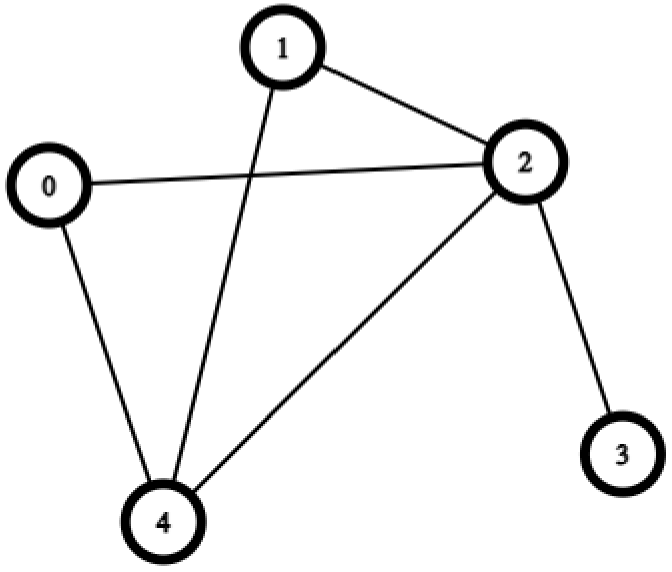
6.2.3 Danh sách kề

MA TRẬN KỀ

Định nghĩa: Với đồ thị có n đỉnh là ma trận vuông cấp n x n có các phần tử 0 và 1. a[i][j], a[j][i] = 1 là một cạnh của đồ thì, = 0 không là một cạnh của đồ thị

ĐỒ THỊ VÔ HƯỚNG

Tính chất: là ma trận đối xứng, tổng các phần tử trên ma trận bằng 2 lần số cạnh, tổng các phần tử trên hàng hoặc cột thứ u là bậc của đỉnh u



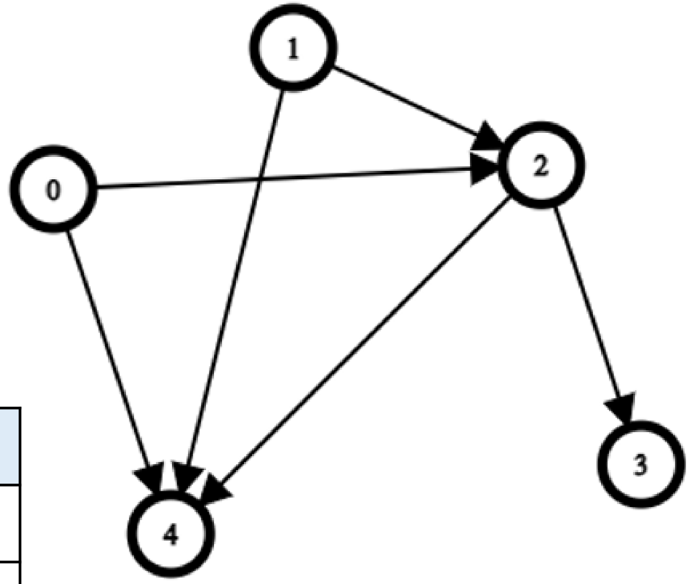
	0	1	2	3	4
0	0	0	1	0	1
1	0	0	1	0	1
2	1	1	0	1	1
3	0	0	1	0	0
4	1	1	1	0	0

ĐỒ THỊ CÓ HƯỚNG

Tính chất:

- Có thể không đối xứng Tổng các phần tử của ma trận bằng số cạnh
- Tổng các phần tử trên hang thứ u là bán bậc ra của đỉnh u
- Tổng các phần tử trên cột thứ u là bán bậc vào của đỉnh u

	0	1	2	3	4
0	0	0	1	0	1
1	0	0	1	0	1
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	0	0	0



ƯU ĐIỂM, NHƯỢC ĐIỂM

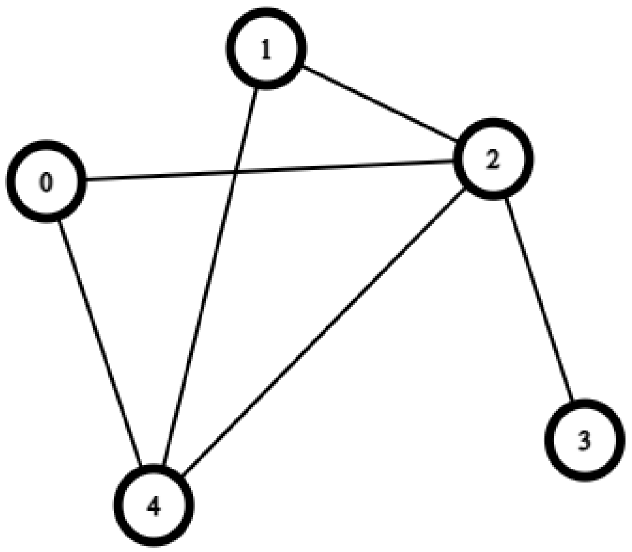
- Ưu điểm: đơn giản, dễ cài đặt, dễ kiểm tra 2 đỉnh kề nhau hay không trong o(1) bằng các kiểm tra giá trị của a[i,j]

- Nhược điểm: tốn bộ nhớ không thể biểu diễn được với đồ thị số đỉnh lớn

DANH SÁCH CẠNH

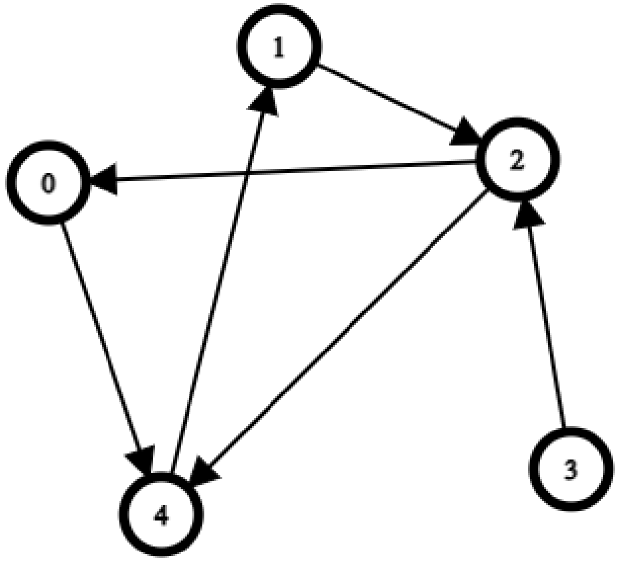
- Thường được biểu diễn khi đồ thị thưa (số lương cạnh $\leq$ 6 lần số đỉnh)
- Đối với đồ thị vô hướng, nếu tồn tại cạnh giữa 2 đỉnh u, v. CHỉ cần liệt kê cạnh (u,v) không cần liệt kê cạnh (v,u) thường chọn u < v, Thường liệt kê các cạnh theo thứ tự tang dần định đầu của các cạnh
- Đối với đồ thị có hướng. Mỗi cạnh là bộ có tính đến thứ tự các đỉnh
- Chú ý: Trong trường hợp đồ thị có hướng, chú ý hướng của cạnh. Với đồ thị có hướng có trọng số ta làm tương tự như với đô thị vô hướng có trọng số

ĐỒ THỊ VÔ HƯỚNG



Đỉnh đầu	Đỉnh cuối
2	4
0	2
0	4
1	2
1	4
2	3

ĐỒ THỊ CÓ HƯỚNG



Đỉnh đầu	Đỉnh cuối
0	4
1	2
2	0
2	4
3	2
4	1

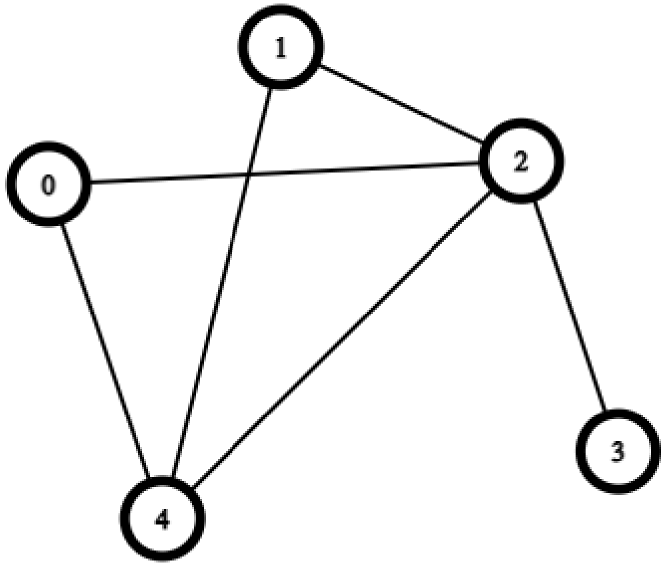
ƯU, NHƯỢC ĐIỂM

- Ưu điểm: tiết kiệm bộ nhớ nếu đồ thị thưa, thuận lời cho các bài toán liên quan tới cạnh của đồ thị
- Nhược điểm: khi cần duyệt các đỉnh kề với đỉnh nào đó bắt buộc phải duyệt tất cả các cạnh dẫn tới chi phí tính toán lớn

DANH SÁCH KỀ

- Đối với mỗi đỉnh u của đồ thị,ta lưu trữ danh sạch các đỉnh kề với đỉnh u. trong c++ để lưu trữ danh sách kề của 1 đỉnh, ta dung 1 vector. Khi đó để lưu trữ bộ toàn bộ danh sách kề của các đỉnh ta dung một mảng các vector
- Cách cài đặt:

```
map<int, vector<int>> Adj;
```



Đỉnh	Danh sách kề
0	{2, 4}
1	{2, 4}
2	{0, 1, 3, 4}
3	{2}
4	{1, 2}

ƯU, NHƯỢC ĐIỂM

- **Ưu điểm:**

+ Dễ dàng duyệt các đỉnh kề của một đỉnh
+ Dễ dàng duyệt các cạnh cảu đồ thị trong mỗi danh sách kề
+ Tối ưu về phương pháp biểu diễn

- **Nhược điểm: khó khăn về mảng lập trình**

VẬN DỤNG

(Graph) Hãy viết chương trình để in ra đồ thị vô hướng biểu diễn dưới dạng danh sách cạnh sang biểu diễn dưới dạng danh sách kề

INPUT:

- Dòng đầu chứa 2 số n và m là số đỉnh và số cạnh của đồ thị
- m dòng tiếp theo mỗi dòng là 2 số u, v biểu diễn cạnh u, v của đồ thị ($1 \leq u, v \leq n$). Các cạnh được liệt kê theo thứ tự tăng dần của các đỉnh đầu

OUPUT:

- In ra danh sách kề, liệt kê theo thứ tự tăng dần của đỉnh

INPUT	OUTPUT
5 7 1 2 1 3 1 4 1 5 2 5 3 4 3 5	1 => 2 3 4 5 2 => 1 5 3 => 1 4 5 4 => 1 3 5 => 1 2 3

6.3: Duyệt đồ thị

NỘI DUNG:

6.3.1 Duyệt theo chiều sâu

6.3.2 Duyệt theo chiều rộng

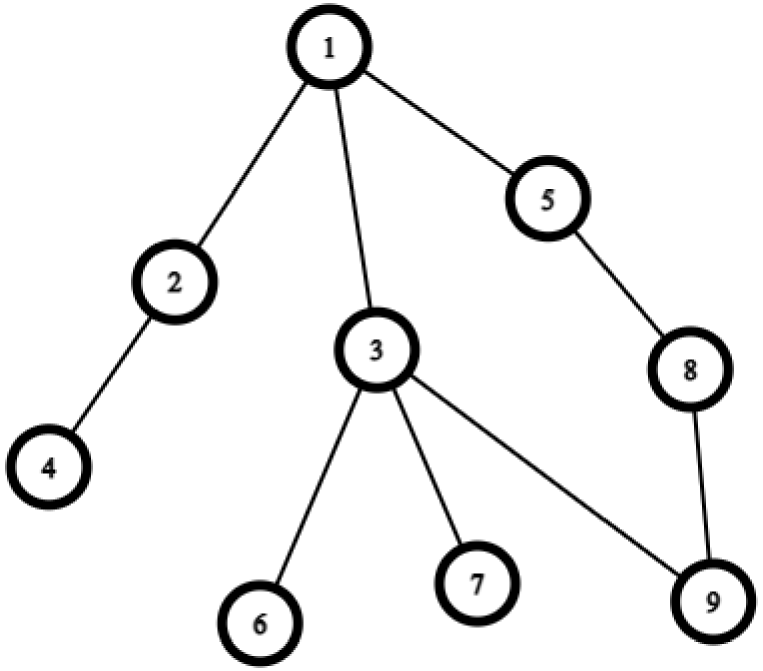
Tìm kiếm theo chiều sâu
(Depth-First Search, viết tắt DFS)

Thuật toán tìm kiếm theo chiều sâu: ưu tiên duyệt xuống nhất có thể trước khi quay lại:

Bắt đầu duyệt từ đỉnh u

```
DFS (u){
        <Xuly Dinh u>;
        visited[u] = true;    // Đánh dấu đỉnh u đã được duyệt qua
        // Duyệt các đỉnh kề với đỉnh u
        for (v : adj[u]){
                if (!visited[v]) {      // Nếu đỉnh v chưa được duyệt
                        DFS(v);    // Đệ quy
                }
        }
}
```

VD:



Kiểm nghiệm thuật toán từ đỉnh 1, trong quá trình mở rộng đỉnh có số thứ tự nhỏ hơn duyệt trước

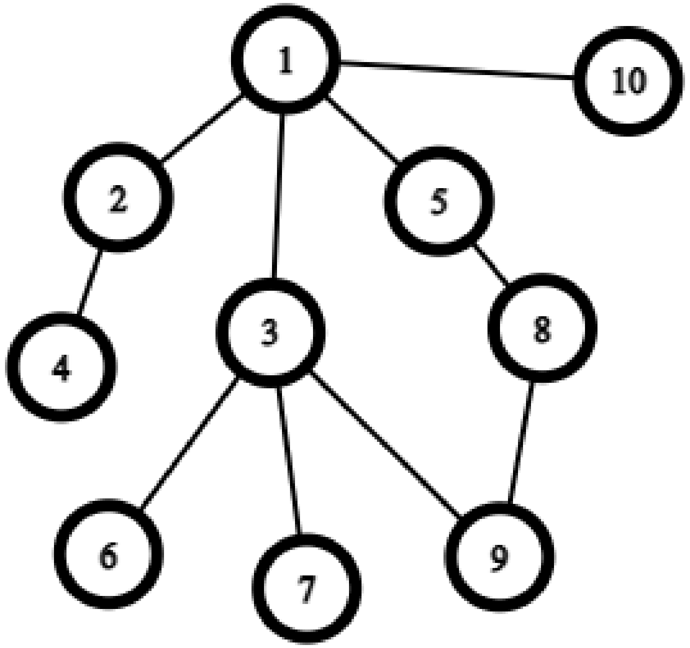
DFS: 1 2 4 3 6 7 9 8 5

Tìm kiếm theo chiều rộng
(Breadth-First Search, viết tắt BFS)

Thuật toán tìm kiếm theo chiều sâu: ưu tiên duyệt duyệt xung quanh trước rồi mới xuống
Bắt đầu duyệt từ đỉnh u

```
BFS (u){
        queue = Ø                  // Tạo một queue rỗng
        push(queue, u) ;           // Đẩy đỉnh u vào hàng đợi
        visited[u] = true;         // Đánh dấu là đỉnh u đã được duyệt
        while (queue != Ø ) {
                v = queue.front()        // Lấy ra đỉnh ở đầu hàng đợi đê kiểm tra
                for (int x : key[u]){    // Duyệt các đỉnh kề với v mà chưa được đẩy vào hàng đợi
                        if (!visited[x]){     // Nếu x chưa được duyệt qua
                                push(queue, x);
                                visited[x] = true;
                        }
                }
        }
}
```

VD:



Kiểm nghiệm thuật toán từ đỉnh 1, trong quá trình mở rộng đỉnh có số thứ tự nhỏ hơn duyệt trước

DFS: 1 2 3 5 10 4 6 7 9 8

ĐỘ PHỨC TẠP

Độ phức tạp của thuật toán phụ thuộc vào cách biểu diễn ma trận:
Đồ thị G = <V, E>

- Biểu diễn bằng ma trận kề: O (V * V)
- Biểu diễn bằng danh sách cạnh: O(V * E)
- Biểu diễn bằng danh sách kề: O(V + E)

SO SÁNH

Tiêu chí	BFS	DFS
Thời gian	$O(b^d)$	$O(b^m)$
Không gian	$O(b^d)$	$O(b * m)$
Hoàn tất	Có	Không (chỉ hoàn thánh khi cây tìm kiếm hữu hạn)
Tối ưu	Có	Không

6.4: Thuật toán tìm đường đi ngắn nhất

NỘI DUNG:

6.1.1 Giới thiệu

6.1.2 Tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại

GIỚI THIỆU

- Áp dụng tìm đường đi ngắn nhất từ 1 đỉnh S tới mọi đỉnh còn lại trên đồ thị có trọng số không âm
- Được công bố lần đầu vào năm 1959
- Được đặt tên của nhà toán học và nhà vật lý người Hà Lan Edsger W.Dijkstra

Ý TƯỞNG

Độ phức tạp: O((E + V)logV)

Ý tưởng cơ bản của thuật toán như sau:

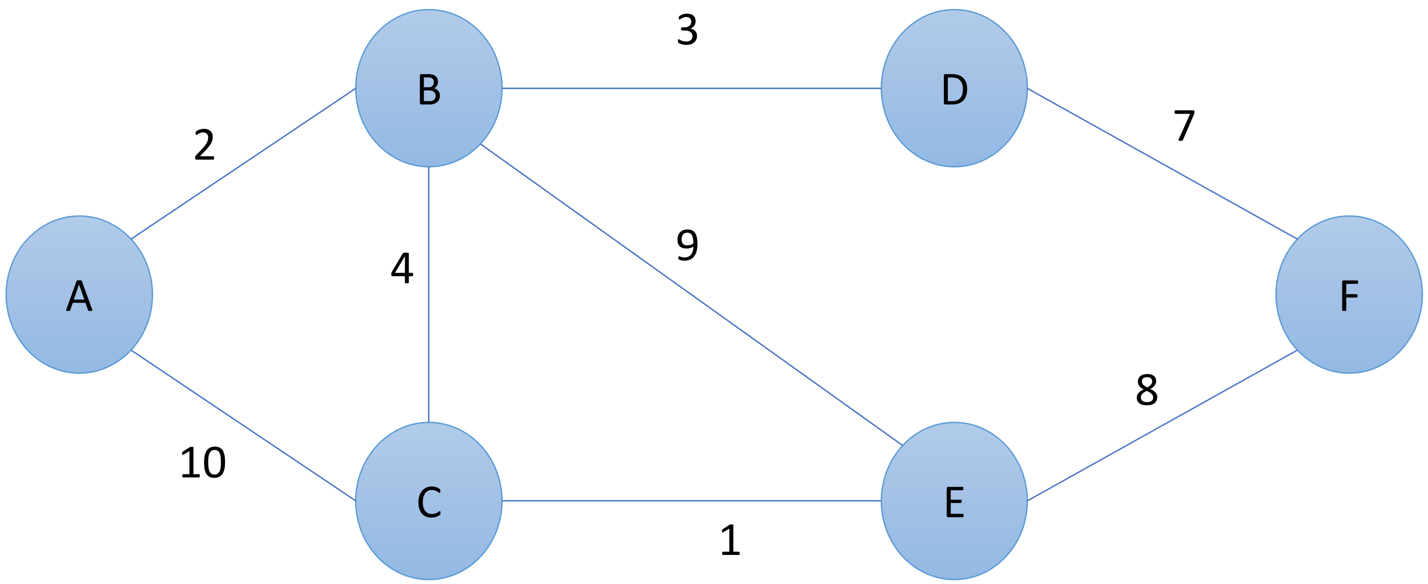
Hàm d(u) dùng để lưu trữ độ dài đường đi (khoảng cách) ngắn nhất từ đỉnh nguồn s đến đỉnh u

$$d(u) = \min\{d(v),\ d(u) + len(u, v), v \in X(u)\}$$

($X(u)$ là tập tất cả các đỉnh có cạnh đi tới đỉnh u)

- Đặt khoảng cách từ đỉnh nguồn s đến chính nó là 0 và đến tất cả các đỉnh khác là vô cùng
- Tiến hành lặp cho đến khi tất cả các đỉnh đã được xác định khoảng cách ngắn nhất từ s hoặc không còn đỉnh nào có thể đạt tới từ s
- Mỗi lần lặp, chọn đỉnh p chưa đi qua có giá trị d(p) nhỏ nhất, cập nhật khoảng cách của các đỉnh kề thông qua đỉnh được chọn p

THUẬT TOÁN DIJKSTRA



THUẬT TOÁN DIJKSTRA

Lặp	Chưa đánh dấu	Đã đánh dấu	A	B	C	D	E	F
	{A, B, C, D, E, F}		0	∞	∞	∞	∞	∞
1	{B, C, D, E, F}	{A}	-	2	10	∞	∞	∞
2	{C, D, E, F}	{A, B}	-	-	6	5	11	∞
3	{C, E, F}	{A, B, D}	-	-	6	-	11	12
4	{E, F}	{A, B, D, C}	-	-	-	-	7	12
5	{E}	{A, B, D, C, E}	-	-	-	-	-	12
6	{}	{A, B, D, C, E}	-	-	-	-	-	-

$$d(u) = \min\{d(v), d\ (u) + len(u,v), v\ \in X(u)\}$$

QUÉT MÃ QR ĐIỂM DANH